

# Minimization of the Reconfiguration Latency for the Mapping of Applications on FPGA-based Systems

Vincenzo Rana  
Politecnico di Milano  
DEI  
Milano, 20133, Italy  
rana@elet.polimi.it

Srinivasan Murali  
Ecole Polytechnique Fédérale  
de Lausanne  
LSI  
Lausanne, 1015, Switzerland  
srinivasan.murali@epfl.ch

David Atienza  
Ecole Polytechnique Fédérale  
de Lausanne  
ESL  
Lausanne, 1015, Switzerland  
david.atienza@epfl.ch

Marco D. Santambrogio  
Massachusetts Institute of  
Technology  
CSAIL  
Cambridge, MA 02139  
santambr@mit.edu

Luca Benini  
Università di Bologna  
DEIS  
Bologna, Italy  
luca.benini@unibo.it

Donatella Sciuto  
Politecnico di Milano  
DEI  
Milano, 20133, Italy  
sciuto@elet.polimi.it

## ABSTRACT

Field-Programmable Gate Arrays (FPGAs) have become promising mapping fabric for the implementation of System-on-Chip (SoC) platforms, due to their large capacity and their enhanced support for dynamic and partial reconfigurability. Design automation support for partial reconfigurability includes several key challenges. In particular, reconfiguration algorithms need to be developed to effectively exploit the available area and run-time reconfiguration support for instantiating at run-time the hardware components needed to execute multiple applications concurrently. These new algorithms must be able to achieve maximum application execution performance at a minimum reconfiguration overhead.

In this work, we propose a novel design flow that minimizes the amount of core reconfigurations needed to map multiple applications dynamically (i.e., using run-time reconfiguration) on FPGAs. This new mapping flow features a multi-stage design optimization algorithm that makes it possible to reduce the reconfiguration latency up to 43%, by taking into account the reconfiguration costs and SoC block reuse between the different applications that need to be executed dynamically on the FPGA. Moreover, we show that the proposed multi-stage optimization algorithm explores a large set of mapping trade-offs, by taking into account the traffic flows for each application, the run-time reconfiguration costs and the number of reconfigurable regions available on the FPGA.

## Categories and Subject Descriptors

C. Computer Systems Organization [C.3 SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS]: Real-time and embedded systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CODES+ISSS'09, October 11–16, 2009, Grenoble, France.  
Copyright 2009 ACM 978-1-60558-628-1/09/10 ...\$10.00.

## General Terms

Algorithms, Design, Performance

## Keywords

Mapping algorithms, Reconfigurable computing, Adaptable computing, FPGAs

## 1. INTRODUCTION

High-end embedded applications, like scalable video rendering and baseband communication protocols, demand a large computation power, while they must meet typical embedded design constraints such as short time-to-market, low energy consumption or reduced implementation size. Embedded execution platforms are thus evolving to become complex *Systems-on-Chip* (SoCs) with lots of post-fabrication flexibility. A common trend in advanced platforms [1] is to couple traditional processing elements (CPUs, DSPs, etc.) and memories with a massive amount of reconfigurable devices, such as *Field-Programmable Gate Arrays* (FPGAs), to enhance the flexibility of the SoCs in different environments [2, 3, 4]. Nevertheless, the capacity of reconfigurable fabrics is always limited (by cost and yield considerations) and one of the most critical areas in application mapping is the definition of efficient dynamic re-configuration strategies for multiple application contexts on limited reconfigurable resources [5, 6].

Within this context we propose a mapping flow that provides optimal dynamic mappings of multiple applications onto FPGAs by exploiting their *Partial Dynamic Reconfiguration* (PDR) capabilities. PDR fully exploits the reconfigurable nature of FPGAs by enabling run-time changes within a portion of the hardware, in order to support a new functionality while the rest of the system keeps running. Thus, the reconfigurable architecture is composed by a static part and by one or several *Reconfigurable Regions* (RR) where the hardware functionalities can be switched in and out according to the demands of the application. This structure helps decreasing the configuration time (i.e., bit-stream downloading) and attempts to ensure continuous functionality of cores mapped to stable regions. Dynamic reconfiguration is a tough problem because of its dual-sided constraints. On the one hand, intra-core communication requirements (bandwidth and latency) need to be accounted

for when mapping applications [7, 8], and cores with critical communication links should be assigned to regions of the configurable fabric that are physically contiguous. On the other hand, the configuration time and energy cost should be minimized to enable fast switching of application contexts under tight timing constraints. Hence, we capitalize on the PDR capabilities of the most advanced FPGAs to address the two aforementioned issues. Given a set of application contexts (groups of applications concurrently executing on a configurable fabrics), the key rationale of our approach is to define mappings that optimally exploit PDR reconfigurability to minimize application context switching cost, while at the same time maximizing communication efficiency in application context execution.

The main contribution of this work is the development of a novel mapping flow that provides efficient dynamic mappings of multiple applications onto FPGAs, by using a multi-stage design optimization algorithm, which takes into account the reconfiguration costs and SoC block reuse between the multiple applications that need to be executed concurrently on the FPGA. Our results show that the obtained designs attain reductions in reconfiguration latency up to 43% with respect to a mapping approach that does not consider the SoC reconfiguration overhead. Moreover, we show in our experimental results that the use of the proposed multi-stage optimization algorithm provides a large set of core mapping trade-offs on the FPGAs (i.e., power, reconfiguration time, SoC performance and area), taking into account the required traffic flows for each application, the reconfiguration cost and the number of reconfigurable regions available on each specific FPGA.

The rest of the paper is structured as follows. In Section 2 we motivate the PDR problem, while in Section 3 we detail the PDR features, key advantages and how it is implemented on Xilinx FPGA devices. In Section 4 we overview previous work on FPGA reconfiguration latency minimization. Then, in Section 5 we introduce our proposed design flow for applications mapping using FPGAs PDR. Next, in Section 6 we summarize the case studies and experimental results obtained in commercial FPGAs. Finally, in Section 7 we draw the main conclusions of this work.

## 2. PROBLEM DESCRIPTION

Multiple applications can be executed on an FPGA and each application can use several soft cores that need to be configured on the device. When an application *A* terminates and a new application *B* is loaded, the cores that only belong to *B* should be loaded in to FPGA. This could be done by reconfiguring the FPGA, possibly removing the cores that only belong to *A*. Aim of this work is to minimize the time required by this reconfiguration process. This goal can be achieved by reconfiguring the minimum number of reconfigurable regions, each one holding an *island* that consists of a certain number of cores, which enables adapting the system to the subsequent application. Let us consider, for instance, applications *A* and *B*, which communication graphs are shown in Figure 1.

The total size of application *A* is 1953 slices, while total size of application *B* is 1996 slices. In order to deploy on the target device both applications *A* and *B*, 2873 slices are needed. If we are using, for instance, a device with 2800 slices, we would be able to configure either application *A* or application *B*, but it would not be possible to deploy both the applications *A* and *B* at the same time on the device. This problem becomes even more relevant when the number of applications increases. Notice that in Figure 1 a few communication arcs (dashed bold lines) are marked as "critical": application performance is strongly impacted by the latency on these channels. Hence, it is highly desirable to map the corre-

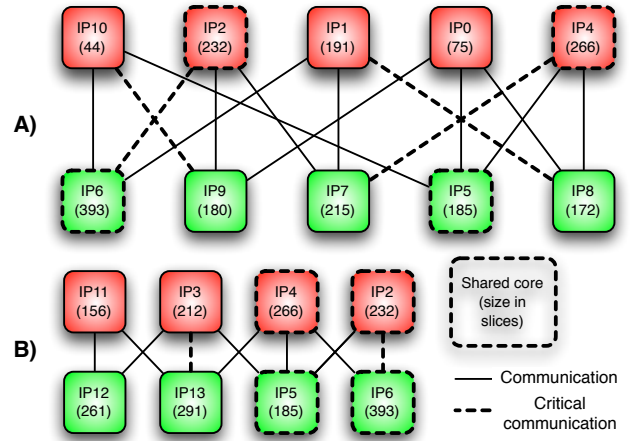


Figure 1: Communication graphs of applications *A* and *B*

sponding cores in the same configuration region, in close physical vicinity, which ultimately implies low-latency communication.

To switch from an application *A* to a *B* one, using a static and complete reconfiguration of a device of 2800 slices, around 155 *ms* would be needed, since the reconfiguration latency strictly depends on the number of slices that have to be reconfigured. To reduce this latency, it is possible to apply partial dynamic reconfiguration techniques, that would enable just configuring a small portion of the device. By applying the proposed design flow to the previously described applications *A* and *B*, utilizing a reconfigurable architecture that consists of 4 reconfigurable regions, it is possible to obtain the output shown in Figure 2. It consists of a set of islands of cores, each one mapped on a single reconfigurable region. In particular, reconfigurable regions 0, 1 and 2 do not need to be configured when passing from application *A* to application *B*, since only region 3 needs to be changed. Thus, the reconfiguration latency is reduced of the 75%, passing from around 155 *ms* to 36 *ms*. Each one of the two configurations is characterized by all the cores needed by either application *A* or application *B*. Moreover, in this simple case the mapping flow succeeded in maintaining all cores linked by critical communication edges within the same configuration region for both applications.

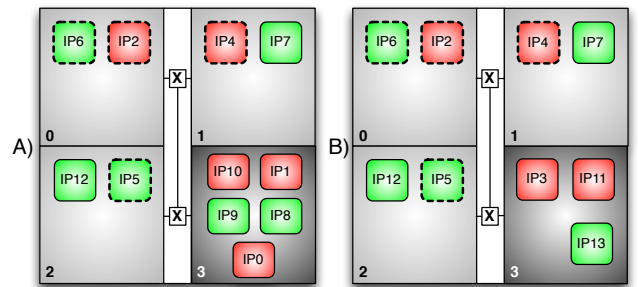
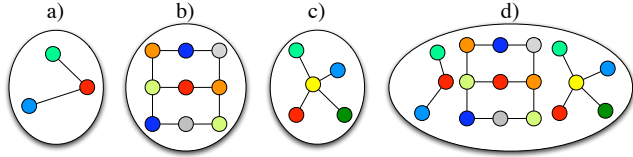


Figure 2: Output of the proposed design flow (for an architecture that consists of 4 reconfigurable regions) for applications *A* and *B*

It is important to note that it is also possible to deploy on the device more than one application (such as the three applications *X*, *Y* and *Z* of Figure 3) at the same time (depending on the size of the reconfigurable device), by simply creating an unconnected communi-

cation graph (e.g., using the communication graphs of applications X, Y and Z in order to create an unconnected communication graph, consisting of three distinct blocks, for a fictitious application V, as shown in Figure 3) and using it every time all the applications have to be deployed together on the target device.



**Figure 3: Communication graphs of applications X (a), Y (b), Z (c) and of the fictitious application V that consists of the three previous applications (d)**

### 3. DYNAMIC RECONFIGURATION

#### 3.1 Dynamic Reconfiguration Characteristics

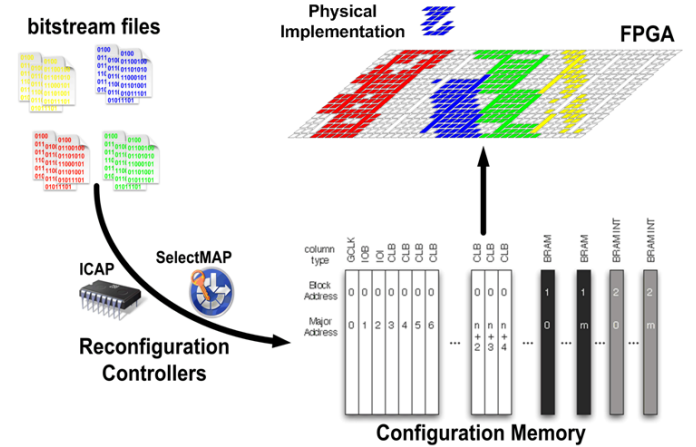
Dynamic reconfiguration and even more, partial dynamic reconfiguration, is one of the key features that make FPGAs unique devices, offering degrees of freedom not available in other akin technologies and in some cases pushing FPGA-based solutions towards the standard application platform e.g., *network controller* capable of handling the TCP and UDP protocols by exploiting partial reconfiguration [9], cryptographic system [10]. In particular, two important benefits can be achieved by exploiting partial dynamic reconfiguration on reconfigurable hardware.

- **The reconfigurable area can be exploited more efficiently with respect to a static design.** There are applications that require a consistent area for the deployment of the system. In this case more reconfigurable resources must be added to the system, or partial dynamic reconfiguration of the available ones can be implemented, in particular if the application can be partitioned into different phases, each building on the previous results, over a certain data set. In this case, the different modules of the application can be configured on the device one after the other, to process data and obtain the final result, while keeping the area requirements lower than having all the functionalities loaded at the same time.
- **Some portion of the application must change over time and react to changes in its environment.** With partial dynamic reconfiguration a portion of the hardware can be adapted over time to cope with new application requirements, giving way to a higher flexibility in system design. In this way *adaptive* systems can be generated, overcoming the inherently static nature of HW solutions.

The two main advantages given by a PDR solution thus address the lack of resources needed to implement an application and its adaptability needs; it must be pointed out that both of the advantages could be replaced by having a larger resource array, where all of the functionalities could be implemented. This solution is not always viable for non-trivial designs and a PDR strategy must be implemented. Reconfigurable hardware taking advantage of partial dynamic reconfiguration can be thus seen as the middle point in the trade-off between the speed of HW solutions and the flexibility of SW.

#### 3.2 Reconfiguration support

In order to configure an FPGA with the desired application, we need to use one or more bitstreams. A bitstream is a binary file in which configuration information for a particular Xilinx device is stored, that is where all the data to be copied on to the configuration SRAM cells, the configuration memory, are stored, along with the proper commands for controlling the chip functionalities. Therefore Virtex devices, such as Virtex II Pro and Virtex 4, are configured by loading application specific data into their configuration memory, as shown in Figure 4.



**Figure 4: Partial dynamic reconfiguration overview and configuration memory setup**

On the Virtex FPGAs the configuration memory is segmented into frames. Virtex devices are partially reconfigurable and a frame is the smallest unit of reconfiguration. According to the device, this element can span the entire length of the FPGA, such as in the Virtex II Pro context, or just part of it, as in Virtex 4 devices. The number of frames and the bits per frame are specific for each device family. The number of frames is proportional to CLB width. Bitstreams can be either partial or full. A full bitstream configures the whole configuration memory and is used for static design or at the beginning of the execution of a dynamic reconfiguration system, to define the initial state of SRAM cells. Partial bitstreams configure only a portion of the device and are one of the end products of any partial reconfiguration flow.

FPGAs provide different means for configuration, under the form of different interfaces to the configuration logic on the chip. There are several modes and interfaces to configure a specific FPGA family, among them the JTAG download cable (which is the method used in this work), the SelectMAP interface, for daisy-chaining the configuration process of multiple FPGAs, configuration loading from PROMs or compact flash cards, microcontroller-based configuration, an internal configuration access port (ICAP) and so on, depending on the specific family. The ICAP provides an interface which can be used by internal logic to reconfigure and read back the configuration memory. In every FPGA a configuration logic is built on the chip, with the purpose of implementing the different interfaces for exchanging configuration data and to interpret the bitstream to configure the device. A set of configuration registers defines the state of this configuration logic at a given moment in time. Configuration registers are the memory where the bitstream file has direct access. Actual configuration data is first written by the bitstream into these registers and then copied by the configuration logic on the configuration SRAMs.

## 4. RELATED WORK

Reconfiguration time minimization can be performed in several ways: the simplest one is the compression of bitstreams [11], while more complex approaches deal with mapping and scheduling techniques, taking into account core prefetching to minimize the number of slices that need to be reconfigured. Bitstream compression can of course be applied on top of mapping and scheduling to further reduce the cost of uploading the reduced-size bitstream.

An optimal algorithm for the minimization of run-time reconfiguration delay has been proposed in [12]. This algorithm is based on the assumption that it is not possible to load at the same time all the cores needed by a single complex application. Thus, the authors try to schedule the tasks of the application in order to reduce the number of partial reconfigurations needed to execute the application itself. The main limits of this approach, in addition to the fact that only a single application is considered, can be found in the assumptions that each task is characterized by both a negligible execution time (according to the reconfiguration time) and the same area on the chip.

The authors of [13] present an approach to handle the dynamic loading of coprocessors in order to decrease the execution time of the applications that are currently running on the system. The main limitation of this approach is that only a single coprocessor can be used for each application. Thus, communication needs among the cores loaded on the FPGA device are not taken into account at all. The approach proposed in our work tries to overcome this limit by defining a communication graph among the cores needed by each application and trying to optimize the communication among them.

Mapping of cores onto Network-on-Chip (NoC) tiles has been explored in many works [7, 14]. In [8], the authors present a methodology for mapping multiple use-cases onto NoCs in order to reduce both area usage and power consumption. However, these works try to find an optimal mapping of the cores, without considering the possibility of dynamically reconfiguring some of the cores.

The problem of packing cores on to the reconfigurable regions is a general case of the bin-packing problem, which has been well studied [15]. As we have a new objective of minimizing the reconfiguration cost, we come up with efficient heuristics to solve the bin-packing problem.

There have been several works on reconfiguration of modules on FPGAs. In [16], the authors present methods to defragment space on the FPGA. An adaptive software/hardware reconfigurable system is presented in [5]. A hardware/software co-synthesis approach for FPGAs is presented in [2]. Adapting the NoC at run-time to suit the application requirements using FPGAs is presented in [3].

Several works have addressed the placement and floorplan issues of reconfigurable FPGAs. In [4], a placement method to reduce reconfiguration data is presented and in [17], the authors present a task placement algorithm. A multi-layer floorplanning of reconfigurable regions is presented in [6].

However, none of the earlier works addresses the important problem of minimizing the amount of reconfigurations when applications running on the FPGA are dynamically changed.

## 5. THE PROPOSED APPROACH

The approach proposed in this paper is based on the design flow depicted in Figure 5, that can be applied on the target reconfigurable architecture described in Section 5.1. It is a design-time approach that consists of three different stages (*Preprocessing*, *Partitioning* and *Mapping*) that are iterated until a feasible solution is found.

The basic idea is to first deploy on the FPGA target device a subset of all the cores needed by all the applications. In order to select (*Selection* phase) the cores of this subset, we use a linear combination (*Ordering* phase) of the size of the cores and their utilization frequency. This set of cores is then partitioned (*Partitioning* phase) into a number of clusters equal to the number of reconfigurable regions. Subsequently, these clusters are mapped on the target architecture (*Primary mapping* phase). Finally, for each application, a subset of cores that are not useful for the selected application is removed in order to free reconfigurable resources for the remaining cores needed by the application itself (*Secondary mapping* phase).

The intuition of adopting a *Preprocessing* phase instead of directly performing the mapping of the cores on the device, as shown in Section 6.6, makes it possible to exploit all the similarity that can be found among the applications, that would have been lost otherwise.

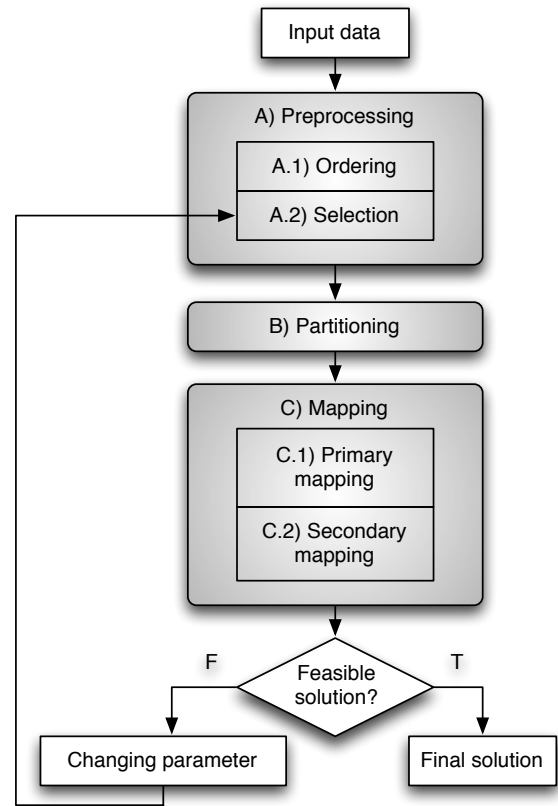


Figure 5: Schema of the proposed design flow

In order to consider also the probability of transition between two applications, it is possible to give a weight to each application in the input set. For instance, if *App0* and *App1* are two applications used very frequently, while *App3* is used very rarely, it is possible to execute the proposed design flow by assigning to *App0* and *App1* a weight of 10, and to *App3* a weight of 1. In this case, the proposed design flow behaves as if 10 applications with the same *communication graph cg* of *App0*, 10 applications with the same *cg* of *App1* and only 1 application with the same *communication graph* of *App3* have to be deployed on the device. In this way, the transition between *App0* and *App1*, that is the most probable one, is optimized with an higher priority in comparison the others (between *App0* and *App3*, and between *App1* and *App3*).



The proposed design flow is based on the following assumptions:

1. all the components of the applications that have to be executed on the target system cannot be completely deployed at the same time on the target device, otherwise no reconfiguration processes are necessary;
2. all the components of each application can be completely deployed on the target device (otherwise the application should be partitioned in several subtasks, each one completely deployable on the target device);
3. the size of each core can not exceed the size of a single reconfigurable region.

## 5.1 Target architecture

The target FPGA-based architecture consists of a set of *static regions* and of a set of *reconfigurable regions*, as shown in Figure 6.

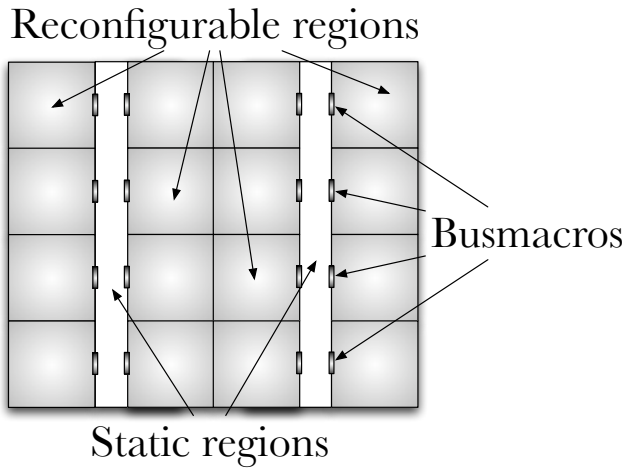


Figure 6: Target architecture schema

The static regions of the architecture are filled with inter-region communication infrastructure (e.g., the switches of a Network-on-Chip optimized for FPGA implementation). The reconfigurable regions are filled up with the cores (both masters and slaves, along with their interfaces towards the communication infrastructure). By following this approach it is possible to completely decouple the inter-region *communication layer* from intra-region *computational layer*. Note that the inter-region communication itself could be considered as reconfigurable. This is beyond the scope of this work, since we are considering a scenario in which the communication infrastructure is fully operational even during reconfiguration processes. Furthermore, all the communication channels between the *static regions* and the *reconfigurable regions* can be guaranteed to be functional, even during reconfiguration processes, by means of *bus-macros*, that are placed along each edge of each *reconfigurable region*, as described in [18] and [19].

## 5.2 Input data

The proposed design flow takes as input:

- a set  $S$  of *communication graphs*, one for each application that has to be deployed on the reconfigurable target architecture,
- the size of the FPGA device ( $DeviceSize$ ),

- the number of reconfigurable regions of the target architecture (*ReconfigurableRegions*).

**DEFINITION 1.** A *communication graph*  $s_z \in S$  is a directed graph,  $G(V, E)$  with each vertex  $v_i \in V$  representing a core and the directed edge  $(v_i, v_j)$ , denoted as  $e_{i,j} \in E$ , representing the communication between the cores  $v_i$  and  $v_j$ . The weight of the vertex  $v_i$ , denoted by  $size_i$ , represents the size of the core in terms of number of slices necessary to configure it on the device. The weight of the edge  $e_{i,j}$ , denoted by  $comm_{i,j}^z$ , represents the sustained rate of traffic flow from  $v_i$  to  $v_j$  weighted by the criticality of the communication.

## 5.3 Preprocessing

### 5.3.1 Ordering

The first step is the ordering of all the cores of the input applications accordingly to a cost metric that takes into account both the size of the selected core and the number of times it appears in the set of applications, as described in Algorithm 1. The size of the core is useful in order to place on the device large cores that could be difficult to place later, while the utilization frequency is needed in order to exploit similarity among the given applications.

Algorithm 1: Ordering (*Set of Applications, g*)

```

1 repeat
2   Select an application A;
3   repeat
4     Select a core C from application A;
5     if ( $C \notin Cores$ ) then
6       add C to Cores;
7       size of C in Cores  $\leftarrow size_c$ ;
8       Instances of C in Cores  $\leftarrow 1$ ;
9     else
10      Instances of C in Cores  $\leftarrow$  Instances of C in
        Cores + 1;
11    end
12  until (no more cores in A);
13 until (no more applications);
14 repeat
15   Select a core C from Cores;
16   CostMetric of C in Cores  $\leftarrow$  Linear combination of
        (Instances of C in Cores) and (size of C);
17 until (no more cores in Cores);
18 Order (Cores);
19 return Cores;
```

### 5.3.2 Selection

The second step makes it possible to select a set  $\gamma$  of cores such as:

$$\sum_{c \in \gamma} size_c \leq \frac{\beta}{100} * DeviceSize \quad (1)$$

This means that the set  $\gamma$  of cores can be completely deployed at the same time on the target device, occupying  $\beta\%$  of the resources available on the device. By using a high value for the  $\beta$  parameter, it is possible to reach a situation where the available resources needed to deploy a core are split over all the reconfigurable regions, making it impossible to configure the core on the target device. For this reason, if the solution generated by the flow with a certain value

of the  $\beta$  parameter is not a feasible one, the design flow is restarted from the *Selection* stage by decreasing the value of the  $\beta$  parameter, until a feasible solution is found, as shown in Figure 5.

Referring to the example presented in Section 2, the set  $\gamma$  contains IP2, IP4, IP5, IP6 (since they are present in both the applications) and IP7, IP12, IP13 (since they are larger than the other remaining cores), accordingly to the metric presented in Section 5.3.1.

## 5.4 Partitioning

The third step is the partitioning of  $\gamma$  into a number of partitions that is equal to the number of reconfigurable regions (*ReconfigurableRegions*) of the target architecture. This partitioning has been performed by providing the *Chaco* partitioner [20] with a *global graph*.

**DEFINITION 2.** A *global graph* is a non directed graph,  $G(V, E)$  with each vertex  $v_i \in V$  representing a subset of  $\gamma$  and the edge  $(v_i, v_j)$ , denoted as  $e_{i,j} \in E$ , representing a connection between at least one core  $c_k \in v_i$  and a core  $c_l \in v_j$ . The weight of the edge  $e_{i,j}$ , denoted by  $edge_{i,j}$ , can be evaluated by using the following formula:

$$edge_{i,j} = \alpha * t + (1 - \alpha) * u \quad (2)$$

where  $u$  is the number of times a core  $c_k \in v_i$  appears in the same communication graph of a core  $c_l \in v_j$ , while:

$$t = \sum_{s_z \in S, c_k \in v_i, c_l \in v_j} comm_{k,l}^z. \quad (3)$$

and  $\alpha$  is the parameter that can be used to reach the desired trade-off between the two previously presented value ( $u$  and  $t$ ).

The *global graph* is initialized by creating a vertex  $v_i$  for each core  $c_j$  of the *Cores* array that has been created in the *Ordering* stage. The *Chaco* partitioner is then used to perform a min-cut on the *global graph*. In particular, a vertex  $v_i \in global\ graph$  can be collapsed with another vertex  $v_j \in global\ graph$  only if:

$$\sum_{c_k \in v_i} size_k + \sum_{c_l \in v_j} size_l \leq \frac{DeviceSize}{ReconfigurableRegions}. \quad (4)$$

The *Partitioning* stage ends when a number of clusters equal to the number of reconfigurable regions have been created.

Referring to the example presented in Section 2, 4 partitions are created.

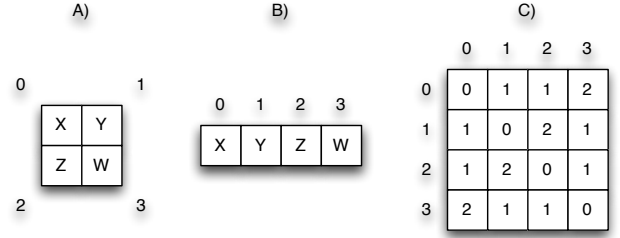
- Partition 1: IP6, IP2 (total size: 625 slices).
- Partition 2: IP4, IP7 (total size: 481 slices).
- Partition 3: IP13 (total size: 291 slices).
- Partition 4: IP12, IP5 (total size: 446 slices).

## 5.5 Mapping

### 5.5.1 Primary mapping

The fourth step of the algorithm is the mapping of the partitions found in the previous step to the actual communication infrastructure architecture. This mapping has been performed with a genetic algorithm developed for this specific problem. Each chromosome has been coded as an array of locations (a single location for each

reconfigurable region) that can be filled up with the identifying marks of the partitions, as shown in Figure 7 (A) and (B) where a simple architecture that consists of 4 reconfigurable regions (0, 1, 2, 3) has been filled with 4 partitions (X, Y, Z, W).

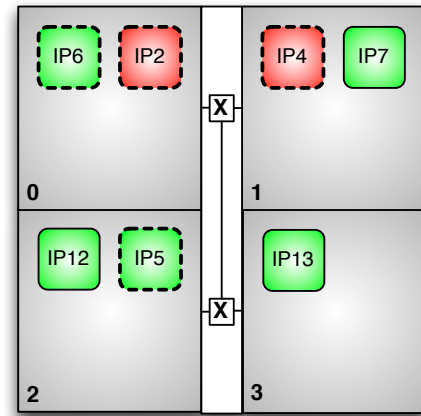


**Figure 7: A simple architecture that consists of a  $2 \times 2$  matrix of reconfigurable regions (A) and the corresponding locations matrix (B). An example of a distance matrix for a  $2 \times 2$  mesh NoC (C)**

The main objective of the genetic algorithm is the minimization of the distance (number of hops) among the partitions that communicate the most. Figure 7 (C) shows an example of the distance matrix, showing the matrix of distances for a  $2 \times 2$  mesh NoC, that can be used to connect the reconfigurable regions presented in Figure 7 (A).

The initial population is generated through several random permutations of the reconfigurable regions on the locations array. The crossover operation has been coded as an operation between two chromosomes that is able to create a new chromosome in which the position of the reconfigurable regions that are located in the same place in both the parents is preserved, while a random permutation is performed on the positions of all the other reconfigurable regions. The mutation operation has been coded as a random swap between two reconfigurable region in a single chromosome. Both the crossover and the mutation operations preserve the validity of all the chromosomes of the population.

Referring to the example presented in Section 2, partition 1 has been mapped onto reconfigurable region 0, partition 2 has been mapped onto reconfigurable region 1, partition 3 has been mapped onto reconfigurable region 3 and partition 4 has been mapped onto reconfigurable region 2, as shown in Figure 8.



**Figure 8: Result of the mapping of the  $\gamma$  set**

### 5.5.2 Secondary mapping

A second mapping phase is necessary in order to find a suitable position even for all the cores that do not belong to  $\gamma$ . Considering each partition that has been placed on the device as an island of cores, it is possible to define the following set of metrics for each island:

- *Size (S)*: this value depends on the selected island and on the current application and represents the percentage of area of the island that can be set as available in the current application; it can be evaluated by subtracting to the size of the reconfigurable region the sum of the areas required by all the cores of the island that belong to the current application;
- *Reconfiguration (R)*: this value depends only on the selected island and represents the percentage of times the island has been selected to be reconfigured; this value ranges from 0% (if all the applications do not need to reconfigure the island) to 100% (if all the applications need to reconfigure the island, since it can be selected to be reconfigured at maximum once for each application). Furthermore, half of the average  $S$  evaluated for the selected island across all the applications is added to this value in order to consider also the probability that the selected island is used as a reconfigurable island by the other application;
- *Communication (C)*: this value depends on the selected island and on the core that has to be mapped and represents the percentage of communication overhead that is generated if the core is mapped on the selected island.

A linear combination of  $S$ ,  $R$  and  $\frac{1}{C}$  is then used in order to select on which island each core not belonging to  $\gamma$  is mapped. The parameter  $\alpha$  can be used to tune the final solution, as described in Section 6, to obtain the desired trade-off between a system in which each application can be deployed after another one with a minimum number of reconfigured regions and a system in which the communication among the islands is minimal. In particular, the average number of reconfigurable regions that have to be reconfigured is evaluated as shown in Figure 9, where *Application 0* needs to configure 2 reconfigurable regions (F and G) of the basic mapping, *Application 1* needs to configure 2 reconfigurable regions (C and G) of the basic mapping and *Application 2* needs to configure 3 reconfigurable regions (E, F and G) of the basic mapping. However, it is not necessary to pass through the basic mapping when switching from one application to another one, as shown in Figure 9: in this case, the average number of regions that have to be reconfigured is

$$\frac{3 + 3 + 4}{3} = 3.33, \quad (5)$$

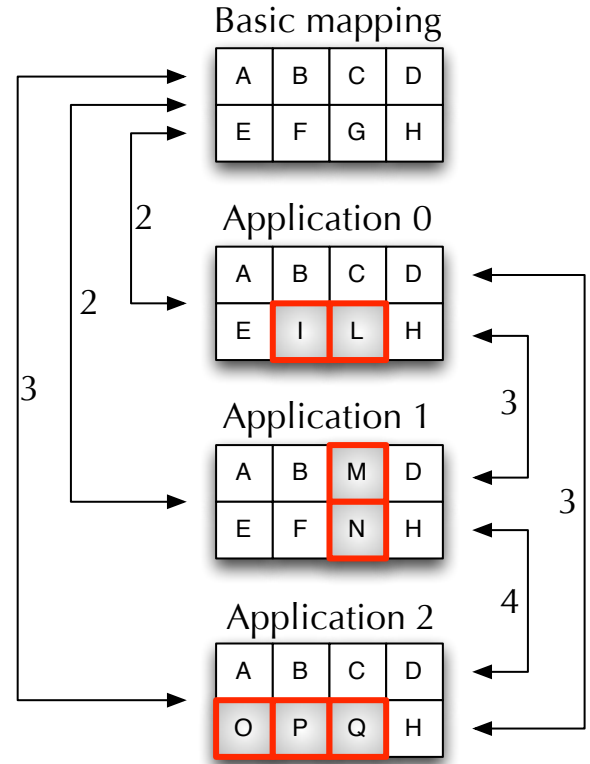
which is 41.67% of the 8 reconfigurable regions.

Referring to the example presented in Section 2, for what concerns application A, IP13 is removed from region 3, in order to place the following cores: IP0, IP1, IP8, IP9 and IP10. On the other hand, for what concerns application B, both IP3 and IP11 are added to IP13 in region 3.

## 6. EXPERIMENTAL RESULTS

### 6.1 Experiments setup

All the experimental results presented in this section, unless differently specified, refer to a target reconfigurable system characterized by 16 reconfigurable regions, provided with a NoC-based



**Figure 9: Evaluation of the number of reconfigurable regions that have to be reconfigured**

inter-tile communication infrastructure [21, 22] and deployed on a Xilinx XC4VLX40 device, that consists of 18432 slices. Since the average execution time of the proposed mapping algorithm is in the order of seconds or tens of seconds, it perfectly fits the timing requirements for a tool used at design-time for the development of a reconfigurable system.

### 6.2 Applications setup

Each experiment presented in this section has been performed, unless differently specified, on a set of 8 different applications (benchmarks of a generic multi-processor system with shared and local memories) that consists of 32 cores. Furthermore, a pool of 64 shared cores has been created and the percentage of cores that resides, for each application, in the pool of shared cores has been set to 50% (even though also 0%, 30% and 70% have been considered in the experiments).

### 6.3 Reconfiguration latency

The reconfiguration latency strictly depends on the number of reconfigurable regions that have to be configured. In particular, Table 1 presents the average reconfigurable latency for a single reconfigurable region on several different Xilinx Virtex 4 devices. The total reconfiguration latency can be obtained by multiplying the latency of the reconfiguration of a single region for the number of regions that have to be configured. It is important to state that reducing the number of reconfigurable regions that have to be reconfigured to switch from an application to another one does not only reduce the time required to switch application, but also makes it possible for the islands that are not involved in the reconfiguration process to continue working. In fact, all the cores belonging to these islands

can go on with their elaboration without any interruption, until they need to communicate with a core that is placed in an island that is being configured on the device. This is another advantage of the proposed approach with respect to a scenario in which a complete reconfiguration is employed.

**Table 1: Reconfiguration latency for a single reconfigurable region on several Xilinx Virtex 4 devices**

Device	Size (slices)	Reconfigurable regions	Reconfiguration latency for a single region (ms)
XC4VLX15	6144	16	21
XC4VLX15	6144	32	10
XC4VLX25	10752	16	38
XC4VLX25	10752	32	18
XC4VLX40	18432	16	64
XC4VLX40	18432	32	33
XC4VLX60	26624	16	93
XC4VLX60	26624	32	46

#### 6.4 Assessment of reconfiguration latency impact

The proposed approach is able to reduce the reconfiguration latency from 15.9% to 43.2% (29.1% in the average) when compared with an approach that does not consider the reconfiguration costs in the mapping phase. This reduction is more pronounced when the number of reconfigurable regions of the system is small or medium (as in real-world scenarios), since the proposed approach is able to take advantage from the packing of the cores shared by more applications in the same reconfigurable region. For instance, the proposed approach can reduce the reconfiguration latency from 482.2 ms to 302.2 ms (37.3% of reduction) in a system with 16 reconfigurable regions (it is important to note that a complete reconfiguration of the device would require more than 1s). This reduction directly impacts the timing performance of the whole system and it is valuable for most of the systems that have to work in real-time. The experimental results related to this comparison, when the number of reconfigurable regions varies from 6 to 32, are shown in Figure 10.

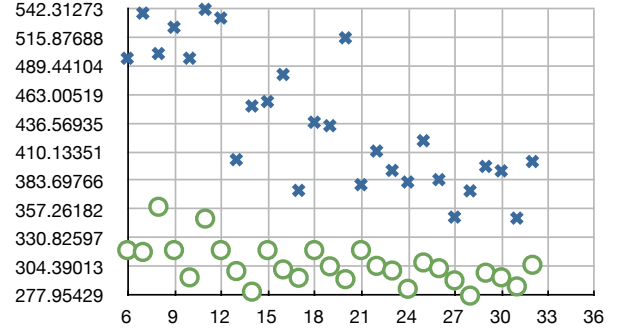
#### 6.5 Exploration of average latency minimization

As shown in Figure 11, the  $\alpha$  parameter can be used to considerably reduce the average number of hops of the final system, thus reducing the average latency of each communication.

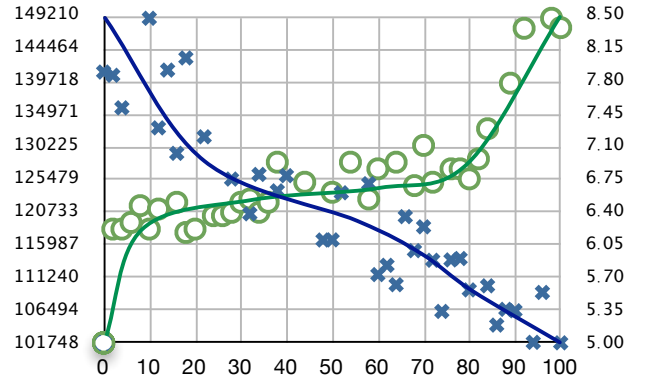
The main drawback of using a high value for the  $\alpha$  parameter is that the average number of reconfigurable regions that have to be reconfigured would be increased, as shown in Figure 11. A value around 70% is usually used in the proposed design flow in order to obtain a good trade-off, even if the designer can manually tune this parameter to achieve the desired solution.

#### 6.6 Impact of the available FPGA reconfigurable area

The  $\beta$  parameter represents the portion of the device that is filled with the partitions generated by the *Chaco* partitioner. If the  $\beta$  parameter is set to 0, all the cores are placed by the *SecondaryMap*-



**Figure 10: The number of reconfigurable regions of the system on the x-axis ranges from 6 to 32, while the y-axis represents the average reconfiguration latency in ms for the proposed approach (o) and for an approach that do not consider reconfiguration costs (x)**



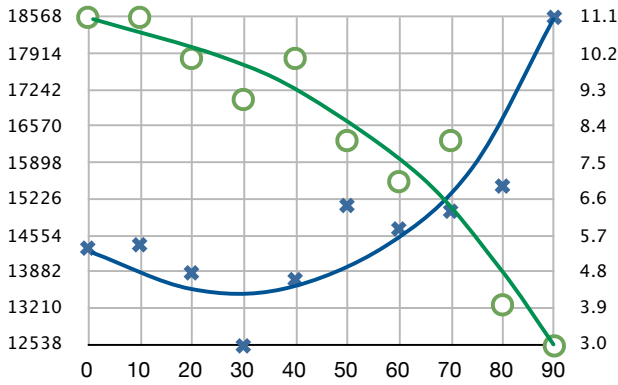
**Figure 11: The  $\alpha$  parameter on the x-axis ranges from 0 to 100, while the y-axis on the left represents the sum of the number of hops multiplied by the rate of the traffic flow on the connection (x) and the y-axis on the right represents the average number of reconfigurable regions that have to be reconfigured (o)**

ping, making impossible to fully exploit all the similarity that can be found among the applications in the partitioning phase. This experiment has been performed by deploying two different applications on a Xilinx XC4VSX35 device.

As shown in Figure 12, using a high value (for instance, 80%) for the  $\beta$  parameter does not substantially affect the global communication metric  $C$  (setting  $\beta$  to 80% lead to increase  $C$  of less than 7% with respect to a scenario in which  $\beta$  is set to 0%). On the other hand, the average number of reconfigurable regions that have to be reconfigured can be reduced from around 11.1 (with  $\beta$  set to 0%) to around 4 (with  $\beta$  set to 80%), with a reduction of around 64%.

Thus, the first iteration of the proposed design flow is performed by setting the  $\beta$  parameter to 80%, while the following ones are performed by assigning to the  $\beta$  parameter decreasing values, until a feasible solution is found (even if the user can also manually tune this parameter).

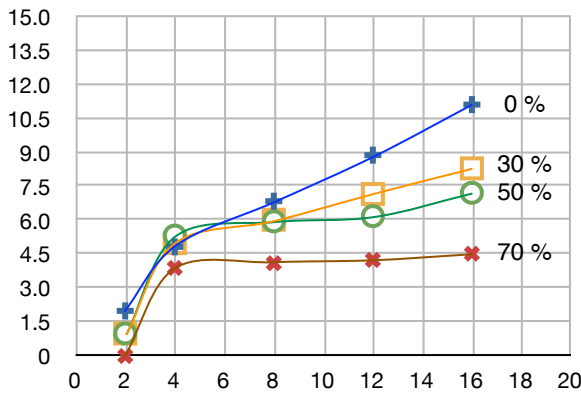




**Figure 12:** The  $\beta$  parameter on the x-axis ranges from 0 to 90, while the y-axis on the left represents the sum of the number of hops multiplied for the rate of the traffic flow on the connection (x) and the y-axis on the right represents the average number of reconfigurable regions that have to be reconfigured (o)

## 6.7 Impact of the number of applications

As it is shown in Figure 13, the average number of reconfigurable regions that have to be reconfigured is almost constant with respect to the number of applications that are deployed on the target system, especially when the percentage of cores that have been selected within the pool of shared cores is quite high (30% or more).



**Figure 13:** The number of applications on the x-axis ranges from 2 to 16, while the y-axis represents the average number of reconfigurable regions that have to be reconfigured, if the percentage of shared cores is set to 0%, 30%, 50% or 70%

Thus, the proposed approach scales very well when the number of applications increases, if they are characterized by enough similarities.

## 7. CONCLUSIONS

Field-Programmable Gate-Arrays (FPGAs) have become a very promising technology in the design of flexible System-on-Chip (SoC) platforms, due to their continuous area capacity increase and capability to dynamically upload new SoC designs and updates, conversely to classical ASIC designs. Furthermore, the exploitation of

FPGA dynamic reconfiguration requires the development of new design flows and algorithms that can efficiently decide how to map the different SoC blocks in the available slots of the underlying communication infrastructure. In this work, we have proposed a novel design flow that can minimize the amount of reconfigurations when dynamic application capabilities are exploited on an FPGA using NoC-based interconnects. This new mapping flow exploits a multi-stage design optimization flow that can achieve reductions in reconfiguration latency up to 43%. In addition to this, it is important to note that all the islands of cores that are not interested in a reconfiguration process, when switching from an application to the subsequent one, can continue working without any interruption. Moreover, we have shown in our experimental results that the use of the proposed multi-stage optimization algorithm enables the exploration of a large set of mapping trade-offs, taking into account the required traffic flows for each application, the reconfiguration cost and the number of reconfigurable regions available on a certain FPGA.

## Acknowledgments

This work was partially supported by the HiPEAC network of excellence ([www.hipeac.net](http://www.hipeac.net)).

## 8. REFERENCES

- [1] Flamand, E.: Strategic directions towards multicore application specific computing. Design, Automation and Test in Europe Conference and Exhibition, 2009 (2009)
- [2] L. Shang, e.a.: Slopes: HardwareSoftware cosynthesis of low-power real-time distributed embedded systems with dynamically reconfigurable fpgas. IEEE TCAD **26** (2007) 508–526
- [3] Hubner, M., Braun, L., Gohringer, D., Becker, J.: Run-time reconfigurable adaptive multilayer network-on-chip for fpga-based systems. Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on (April 2008) 1–6
- [4] Chen, W., Wang, Y., Wang, X., Peng, C.: A new placement approach to minimizing fpga reconfiguration data. Embedded Software and Systems, 2008. ICESS '08. International Conference on (July 2008) 169–174
- [5] Becker, J., Hubner, M., Hettich, G., Constapel, R., Eisenmann, J., Luka, J.: Dynamic and partial fpga exploitation. Proceedings of the IEEE **95**(2) (Feb. 2007) 438–452
- [6] Singhal, L., Bozorgzadeh, E.: Multi-layer floorplanning on a sequence of reconfigurable designs. Field Programmable Logic and Applications, 2006. FPL '06. International Conference on (Aug. 2006) 1–8
- [7] Murali, S., De Micheli, G.: Bandwidth-constrained mapping of cores onto noc architectures. Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings **2** (Feb. 2004) 896–901 Vol.2
- [8] Murali, S., Coenen, M., Radulescu, A., Goossens, K., De Micheli, G.: A methodology for mapping multiple use-cases onto networks on chips. Design, Automation and Test in Europe, 2006. DATE '06. Proceedings **1** (March 2006) 1–6
- [9] Chaubal, A.P.: Design and implementation of an fpga-based partially reconfigurable network controller. Master's thesis, Virginia Polytechnic Institute and State University (2004)
- [10] Castillo, J., Huerta, P., López, V., Martínez, J.I.: A secure self-reconfiguring architecture based on open-source

- hardware. In: International Conference on Reconfigurable Computing and FPGAs. (2005)
- [11] Ju Hwa Pan Mitra, T.W.F.W.: Configuration bitstream compression for dynamically reconfigurable fpgas. Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on (November 2004) 766–773
  - [12] Ghiasi, S., Nahapetian, A., Sarrafzadeh, M.: An optimal algorithm for minimizing run-time reconfiguration delay. Trans. on Embedded Computing Sys. **3**(2) (2004) 237–256
  - [13] Huang, C., Vahid, F.: Dynamic coprocessor management for fpga-enhanced compute platforms. In: CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems, New York, NY, USA, ACM (2008) 71–78
  - [14] A. Hansson, e.a.: A unified approach to mapping and routing on a combined guaranteed service and best-effort network-on-chip architectures. Technical Report No: 2005/00340, Philips Research (2005)
  - [15] Vazirani, V.: Approximation algorithms. Springer-Verlag (2001)
  - [16] Fekete, S., van der Veen, J., Ahmadiania, A., Gohringer, D., Majer, M., Teich, J.: Offline and online aspects of defragmenting the module layout of a partially reconfigurable device. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **16**(9) (Sept. 2008) 1210–1219
  - [17] Lu, Y., Marconi, T., Gaydadjiev, G., Bertels, K., Meeuws, R.: A self-adaptive on-line task placement algorithm for partially reconfigurable systems. Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on (April 2008) 1–8
  - [18] Xilinx: Xapp290 - two flows for partial reconfiguration: Module based or difference based (September 2004)
  - [19] Xilinx: Early access partial reconfiguration user guide. (March 2006)
  - [20] Hendrickson, B., Leland, R.W.: A multi-level algorithm for partitioning graphs. In: Supercomputing. (1995)
  - [21] Dall'Osso, M., Biccari, G., Giovannini, L., Bertozzi, D., Benini, L.: Xpipes: a latency insensitive parameterized network-on-chip architecture for multiprocessor socs. Computer Design, 2003. Proceedings. 21st International Conference on (Oct. 2003) 536–539
  - [22] Bertozzi, D., Benini, L.: Xpipes: a network-on-chip architecture for gigascale systems-on-chip. Circuits and Systems Magazine, IEEE **4**(2) (2004) 18–31